ソフトウェア処理方法およびソフトウェア処理システム

BACKGROUND OF THE INVENTION

本発明は、プロセッサと共有リソースを有するシステムにかかわり、特には、共有リソース競合時の処理速度の低下を防止する技術に関するものである。

例えばＣＰＵのようなプロセッサとＤＳＰ（デジタルシグナルプロセッサ）で構成されるマルチプロセッサシステムにおいては、処理をプロセッサとＤＳＰで分担して実行することにより処理性能の向上を図る。メモリ、バス、周辺回路などを共有リソースとして共用することにより、チップ面積を縮小し、低コストを実現する。

プロセッサが外部メモリへのデータ読み出し中に、ＤＳＰからも同一のメモリに対してデータ読み出しの処理が発行され、アクセス競合が発生した場合、バス調停回路によりアクセス競合に対する処理が制御される。バス調停回路の制御方法としては、ＤＳＰからのデータ読み出し処理は受け付けるが、プロセッサからのデータ読み出しが終了するまではＤＳＰからのデータ読み出しの処理は待たせる。

また、プロセッサとＤＳＰとで構成され、プロセッサがＤＳＰに対して突き放し制御するマルチプロセッサシステムにおいて、ＤＳＰはプロセッサからの処理要求をＤＳＰの状態に関係なく受け付けるが、ＤＳＰが別の処理を実行中であれば、すぐには処理を開始せずにしばらく待ち、実行できる状態になった時点で自律的に処理を実行し、処理結果をプロセッサ側へ返す。

しかしながら、従来の共有リソースを有するシステムでは、共有リソースへのアクセス競合が頻繁に発生するような場合、先の処理が終了するまで待つ頻度が高くなり、システム全体の処理速度が低下する要因となる。

さらに、調停回路は共有リソースの状態のみを把握しており、システム全体の状態を把握していない。例えばＤＳＰが他の処理を実行中にプロセッサからＤＳＰへの処理要求が発行された場合、従来の調停回路では、先の処理が終了するまでその処理を待たせる。一方、プロセッサはＤＳＰへ発行した処理の結果を受けて次の処理を実行するため、プロセッサも待ち状態となる場合がある。このような状況では、先の処理が終了するまでＤＳＰへ発行された処理およびその発行元

であるプロセッサが待ち状態となり、プロセッサおよびＤＳＰを効率良く使用することができない。

また、上記従来の共有リソースを有するシステムでは、ＤＳＰへの処理要求が重複するＤＳＰ競合を避けるためにソフトウェアをＣＰＵとＤＳＰに適切に割り振る必要がある。この割り振り作業は人手で行っており、ソフトウェアの修正が発生する毎に多大な時間を必要とし、工数がかかるという問題がある。

SUMMARY OF THE INVENTION

したがって、本発明の主たる目的は、リソース競合が発生した場合に、できるだけリソース競合時の処理の停止を避け、システムの処理速度の低下を抑制できるソフトウェア処理方法・システムを提供することである。

本発明の他の目的、特徴、利点は、後述から明かになるであろう。

以下、複数種類の構成要素手段が記述されるが、これら各手段については、ソフトウェアで構成してもよいし、あるいはハードウェアで構成してもよいし、あるいはハードウエアとソフトウエアとの組み合わせで構成してもよい。

上記の目的を達成するために、本発明は次のような手段を講じる。

本発明によるソフトウェア処理方法は、プロセッサとプロセッサが使用するリソースで構成されるシステムに対し、プロセッサが使用するリソースの使用状況を監視する使用状況の監視処理の過程と、前記使用状況の監視処理の過程で得られる競合情報に応じて、実行されるソフトウェアの処理方法を適応的に変更するソフトウェア処理の変更処理の過程とを含むものである。

上記のソフトウェア処理方法に対応する本発明によるソフトウェア処理システムは、プロセッサと、前記プロセッサが使用するリソースと、前記リソースの使用状況を監視する使用状況の監視装置と、前記使用状況の監視装置で得られる競合情報に応じて、実行されるソフトウェアの処理方法を適応的に変更するソフトウェア処理の変更装置を有するものである。

上記構成によれば、リソースの競合を動的に判定し、競合頻度がある程度以上に高い場合には、リソースを使用するソフトウェアの処理方法を適応的に変更することによりリソース競合時の処理の停止を避け、システムの処理速度の低下を抑制することができる。

上記において、前記ソフトウェア処理の変更処理の過程、または、前記ソフトウェア処理の変更装置についての１つの態様を、ソフトウェアがある処理を行うのに複数の実行方法を持ち、前記ソフトウェアの実行中に前記使用状況の監視処理の過程で得られる競合情報に応じて、前記複数の実行方法の中から１つを選択するものとする。複数の実行方法の中からの選択であり、最も簡単な方式である。

上記において、リソースが、処理の記憶装置である場合において、前記使用状況監視処理の過程を、前記記憶装置の使用状況を監視するものとする。これによれば、記憶装置へのアクセス競合の発生を軽減し、記憶装置を効率良く使用することが可能となる。したがって、記憶装置の競合の発生によるシステムの処理速度の低下の影響を軽減することができる。

上記において、前期使用状況の監視処理の過程についての１つの態様として、複数クロック遡った記憶装置の使用状況を記憶し、過去および現在の使用状況から前記競合情報を生成するものとする。また、上記において、前記使用状況の監視処理の過程についての別の態様として、前記記憶装置の使用時における使用時間を記憶し、前記使用時間が所定値以上か否かに基づき前記競合情報を生成するものとする。クロック数または時間を指標にして、記憶装置の使用状況を容易に判定することができる。

上記において、前記リソースが、処理の記憶装置、および、前記プロセッサと前記記憶装置を接続するバスである場合において、前記使用状況の監視処理の過程についての１つの態様を、前記バスの使用状況を監視するものとする。この場合、リソースの競合判定をバスアクセスの競合判定で行う。これによれば、バスアクセス競合の発生を軽減し、バスを効率良く使用することが可能となる。したがって、バスアクセスの競合の発生によるシステムの処理速度の低下の影響を軽減することができる。

上記において、前記使用状況の監視処理の過程についての１つの態様として、複数クロック遡ったバスの使用状況を記憶し、過去および現在の使用状況から前記競合情報を生成するものとする。また、上記において、前記使用状況の監視処理の過程についての別の態様として、前記バスの使用時における使用時間を記憶し、前記使用時間が所定値以上か否かに基づき前記競合情報を生成するものとする。クロック数または時間を指標にして、バスの使用状況を容易に判定すること

ができる。

上記において、前記リソースが、前記プロセッサ（これを第１のプロセッサとする）からの処理要求に応じて処理を行う第２のプロセッサである場合において、前記使用状況の監視処理の過程、または、前記使用状況の監視装置についての１つの態様として、前記第２のプロセッサの使用状況の監視を行うものとする。

第１のプロセッサが第２のプロセッサを突き放し制御方式で制御する場合に好適である。突き放し制御方式では、第１のプロセッサが第２のプロセッサに対して処理の要求を行い、第２のプロセッサは自己の状態のいかんにかかわりなく処理要求を受け付け、第１のプロセッサは別の処理の実行に移行する。処理要求を受け付けた第２のプロセッサは、自己の状態を判断し、別の処理を実行中であれば待機し、前記別の処理の実行が完了した後に処理の実行を開始し、処理結果を第１のプロセッサへ返す。この制御方式では、複数の処理が第２のプロセッサで競合することから、第２のプロセッサが共有リソースとなる。

この場合、第２のプロセッサを使用しているときに別の処理が発生した場合、前記競合情報として前記第１のプロセッサへの割り込み信号を用いて、割り込みを発生させる割り込み機能を備えるとよい。割り込みルーチンを用意する以外は、もともとのソフトウェアに対して処理の変更、追加をする必要が無く、ソフトウェアをそのまま適用でき、さらにソフトウェアによる競合対策の処理がないため、ソフトウェアのオーバーヘッドが少ない。

上記のソフトウェア処理方法が、さらに、同一アドレスでアクセス可能な複数のメモリバンクを有する場合において、前記使用状況の監視処理の過程で得られる競合情報についての１つの態様として、前記複数のメモリバンクのうち１つのメモリバンクの選択を示す信号を用いることにする。この場合、第２のプロセッサでの処理の実行毎に、あるいは第２のプロセッサで実行された処理が終了する毎に、複数のメモリバンクの中から１つを選択する選択信号を切り替える機能を備えるとよい。

メモリバンクを選択することで、プロセッサ競合時の処理の停止を避け、システムの処理速度の低下を抑制することができる。競合対策の処理がメモリバンク切り替え回路のみで実現できるため、他の方法に比べて高速である。

上記のソフトウェア処理方法が、さらにソフトウェアのコンパイラを有する場

合において、前記コンパイラについての１つの態様として、次のように構成する。すなわち、

ソフトウェアから前記リソースを使用する処理であるか否かを識別する処理識別手段と、

前記処理識別手段により識別された処理と等価な、前記リソースを使用しない等価処理と、

前記使用状況の監視処理の過程で得られる競合情報により使用状況を判定する使用状況の判定処理と、

前記使用状況の判定処理の結果に応じて、前記処理識別手段により識別された処理を適応的に前記等価処理に置き換える置換処理と

を、前記ソフトウェアに追加する。これにより、リソース競合を避け、システムの処理速度の低下を避ける。

また、上記のソフトウェア処理方法が、さらにソフトウェアのコンパイラを有する場合において、前記コンパイラについての別の１つの態様として、次のように構成する。すなわち、

ソフトウェアから前記リソースを使用する処理であるか否かを識別する処理識別手段と、

前記処理識別手段により識別された処理と等価な、前記リソースを使用しない等価処理と、

現時刻の前記使用状況の監視処理の過程で得られる競合情報を記憶する記憶処理と、

過去の時刻における前記競合情報により使用状況を判定する使用状況の判定処理と、

前記使用状況の判定処理の結果に応じて、前記処理識別手段により識別された処理を適応的に前記等価処理に置き換える置換処理と

を、前記ソフトウェアに追加する。これによれば、競合の履歴を加味した処理となり、リソース競合の判定を高精度に行うことができる。

上記において、前記競合情報についての１つの態様を、前記リソースの処理要求が発行されてから処理が終了するまでの「処理時間」とする。この場合に、前記使用状況の判定処理は、前記処理時間をある設定値と比較する処理を行うもの

である。

また、上記において、前記競合情報についての別の態様を、前記リソースの処理要求が発行されてから処理が実行するまでの「待ち時間」とする。この場合に、前記使用状況の判定処理は、前記待ち時間をある設定値と比較する処理を行うものである。

これによれば、処理の待ち時間や処理時間などを計測することでリソース競合が発生中か否かを判定し、リソース競合が発生中ならば新たにリソース競合を引き起こす処理を発行しない。これにより、競合によるシステムの処理速度の低下を抑制することが可能である。

上記において、前記使用状況の判定処理についての１つの態様として、定期的または不定期に前記リソースの使用状況の判定を見直すものとする。この場合に、乱数を用いて前記リソースの使用状況の判定を見直すのがよい。これによれば、定期的または不定期に競合判定を補正するので、さらに高精度な競合判定が可能である。

上記のソフトウェア処理方法において、前記コンパイラにおける前記処理識別手段により抽出される処理が前記ソフトウェアの複数の箇所から抽出される場合、前記コンパイラについての１つの態様として、次のように構成するものとする。すなわち、さらに、前記処理識別手段で識別された処理の出現箇所を識別する出現箇所の識別処理を前記ソフトウェアに追加し、前記記憶処理は前記出現箇所毎に前記競合情報を記憶し、前記使用状況の判定処理は前記出現箇所毎に記憶した前記競合情報を用いて判定を行うことである。これによれば、競合情報を競合を引き起こす処理の出現箇所とともに記憶することで、ループ処理における競合判定を高精度に行うことが可能である。

以上のようにして、本発明によれば、リソースの競合およびプロセッサ上で実行する処理の競合を考慮し、競合が発生した場合にはできるかぎり処理の停止を避け、システムの処理速度の低下を抑制することができる。また、処理の割り当てを人手で行う必要がなくなり、工数の削減が図れる。

The foregoing and other aspects will become apparent from the following description of the invention when considered in conjunction with the accompanying drawing figures.

BRIEF DESCRIPTION OF THE DRAWING FIGURES

ＦＩＧ．１は本発明の実施の形態におけるソフトウェア処理システムの基本構成を示すブロック図、

ＦＩＧ．２は本発明の第１の実施の形態におけるソフトウェア処理システムのハードウェアの構成を示すブロック図、

ＦＩＧ．３は本発明の第１の実施の形態におけるソフトウェア処理システムのバスの使用状況の監視装置の構成を示す図、

ＦＩＧ．４Ａは本発明の第１の実施の形態におけるソフトウェア処理システムのソフトウェアの構成を示す図、ＦＩＧ．４Ｂはプログラム中で記述された処理を示す図、ＦＩＧ．４Ｃはバスを使用しない等価処理を示す図、

ＦＩＧ．５は本発明の第１の実施の形態におけるソフトウェア処理システムにおけるコンパイルフローを示すフローチャート、

ＦＩＧ．６は本発明の第１の実施の形態におけるソフトウェア処理システムにおけるバスアクセスの識別装置が識別するバスアクセスの状態を示す図、

ＦＩＧ．７Ａ、ＦＩＧ．７Ｂは本発明の第１の実施の形態におけるソフトウェア処理システムにおける処理付加手段によるソフトウェアの変更を示す図、

ＦＩＧ．８は本発明の第２の実施の形態におけるソフトウェア処理システムのハードウェアの構成を示すブロック図、

ＦＩＧ．９は本発明の第２の実施の形態におけるソフトウェア処理システムのバス調停回路の制御フローを示すフローチャート、

ＦＩＧ．１０は本発明の第２の実施の形態におけるソフトウェア処理システムのコンパイラの処理フローを示すフローチャート、

ＦＩＧ．１１Ａ、ＦＩＧ．１１Ｂは本発明の第２の実施の形態におけるソフトウェア処理システムの外部メモリへ頻繁にアクセスする処理に対する変更を示す図、

ＦＩＧ．１２は本発明の第２の実施の形態におけるソフトウェア処理システムのバス競合の判定フローを示すフローチャート、

ＦＩＧ．１３は本発明の第２の実施の形態におけるソフトウェア処理システムの等価処理への置換を行う確率を示す図、

ＦＩＧ．１４は本発明の第２の実施の形態におけるソフトウェア処理システムの出現箇所を考慮した場合のバス競合の判定フローを示すフローチャート、

ＦＩＧ．１５は本発明の第２の実施の形態におけるソフトウェア処理システムのバス競合の判定フローを示すフローチャート、

ＦＩＧ．１６は本発明の第３の実施の形態におけるソフトウェア処理システムのハードウェア構成を示すブロック図、

ＦＩＧ．１７は本発明の第３の実施の形態におけるソフトウェア処理システムのコンパイラの処理フローを示すフローチャート、

ＦＩＧ．１８Ａ、ＦＩＧ．１８Ｂは本発明の第３の実施の形態におけるソフトウェア処理システムのコンパイラによる処理の変更を示す図、

ＦＩＧ．１９は本発明の第３の実施の形態におけるソフトウェア処理システムのＤＳＰ競合の判定フローを示すフローチャート、

ＦＩＧ．２０Ａ、ＦＩＧ．２０Ｂは本発明の第３の実施の形態におけるソフトウェア処理システムの処理の流れを示すシーケンス図、

ＦＩＧ．２１は本発明の第４の実施の形態におけるソフトウェア処理システムのハードウェア構成を示すブロック図、

ＦＩＧ．２２は本発明の第４の実施の形態におけるソフトウェア処理システムの調停回路の制御フローを示すフローチャート、

ＦＩＧ．２３は本発明の第４の実施の形態におけるソフトウェア処理システムのコンパイラの処理フローを示すフローチャート、

ＦＩＧ．２４Ａ、ＦＩＧ．２４Ｂは本発明の第４の実施の形態におけるソフトウェア処理システムのＤＳＰで実行する処理に対する変更を示す図、

ＦＩＧ．２５は本発明の第４の実施の形態におけるソフトウェア処理システムのＤＳＰ競合の判定フローを示すフローチャート、

ＦＩＧ．２６は本発明の第４の実施の形態におけるソフトウェア処理システムの等価処理への置換を行う確率を示す図、

ＦＩＧ．２７は本発明の第４の実施の形態におけるソフトウェア処理システムの出現箇所を考慮した場合のＤＳＰ競合の判定フローを示すフローチャート、

ＦＩＧ．２８は本発明の第４の実施の形態におけるソフトウェア処理システムのＤＳＰ競合の判定フローを示すフローチャート、

ＦＩＧ．２９は本発明の第５の実施の形態におけるソフトウェア処理システムのハードウェア構成を示すブロック図、

ＦＩＧ．３０は本発明の第５の実施の形態におけるソフトウェア処理システムのソフトウェアを示す図、

ＦＩＧ．３１は本発明の第５の実施の形態におけるソフトウェア処理システムのＤＳＰ競合時の処理フローを示すフローチャート、

ＦＩＧ．３２Ａ、ＦＩＧ．３２Ｂは本発明の第５の実施の形態におけるソフトウェア処理システムにおける処理の流れを示すシーケンス図、

ＦＩＧ．３３は本発明の第６の実施の形態におけるソフトウェア処理システムのハードウェア構成を示すブロック図、

ＦＩＧ．３４Ａは本発明の第６の実施の形態におけるソフトウェア処理システムのソフトウェアの構成を示す図、ＦＩＧ．３４Ｂはプログラム中で記述されたＤＳＰを使用するコマンドを含んでライブラリ化された処理を示す図、ＦＩＧ．３４ＣはＣＰＵを使用するコマンドを含んでライブラリ化された等価処理を示す図、

ＦＩＧ．３５は本発明の第６の実施の形態におけるソフトウェア処理システムのコンパイルフローを示すフローチャート、

ＦＩＧ．３６は本発明の第６の実施の形態におけるソフトウェア処理システムのメモリバンクマッピングを示す図である。

In all these figures, like components are indicated by the same numerals.

DETAILED DESCRIPTION

（基本構成）

本発明の実施の形態にかかわるソフトウェア処理システムの基本構成をＦＩＧ．１に示す。

プロセッサ１０１がソフトウェア処理の実行を始めると、使用状況の監視処理の手段１０３は、プロセッサ１０１が使用するリソース１０２の使用状況を監視し、その出力を受けて、リソース１０２の使用状況の情報を取得する。ソフトウェア処理の変更処理の手段１０４は、リソース１０２の使用状況の情報の結果に

応じて、実行されるソフトウェアの処理方法を適応的に変更する。

なお、使用状況の監視処理の手段１０３およびソフトウェア処理の変更処理の手段１０４については、それぞれソフトウェアで実現するものを想定しているが、これのみに限定する必要はなく、それぞれハードウェアで実現しても良い。少なくとも、リソース１０２の使用状況を監視し、使用状況の情報を取得する機能、および、使用状況の情報の出力を受けてソフトウェアの処理方法を適応的に変更する機能を有していれば良い。

以下、本発明の第１から第６までの実施の形態について、図を用いて説明する。

（第１の実施の形態）

以下、本発明の第１の実施の形態を図面に基づいて説明する。

ＦＩＧ．２は、第１の実施の形態にかかわるソフトウェア処理システムのハードウェアの構成を示すブロック図である。１００１はＣＰＵ、１００２はＣＰＵ１００１が使用するバス、１００３はＣＰＵ１００１の周辺回路、１００４は外部メモリ、１００５はバス１００２上のバスアクセスの状態を記憶する使用状況の監視装置である。

ＦＩＧ．３は、前記使用状況の監視装置１００５の構成を示す図である。１１０１はバス１００２上でバスアクセスが起きているかどうかを特定の期間だけ識別するバスアクセスの識別装置、１１０２はバスアクセスの識別装置１１０１の出力を受けて、バスアクセスの状態を記録するバスアクセスの状態レジスタである。

ＦＩＧ．４は、前記ソフトウェア処理システムのソフトウェアの構成を示す図である。１２０１はＣＰＵ１００１上で動作するプログラム、１２０２，１２０３，１２０４はプログラム１２０１中に記述された処理、１２０５，１２０６，１２０７は処理１２０２，１２０３，１２０４とそれぞれ同等の機能を持ち、バス１００２を使用しないＣＰＵ１００１用の処理である。

ＦＩＧ．５は、前記ソフトウェア処理システムにおけるコンパイルフローを示すフローチャートである。１３０１はコンパイラがプログラム１２０１をコンパイルするコンパイル手段である。１３０２はプログラム１２０１に記述された処理において、バス１００２を使用する処理であるか否かを識別する処理識別手段

である。１３０３は使用状況の判定処理と置換処理を付加する処理付加手段である。ここで使用状況の判定処理とは、処理識別手段１３０２の出力を受けて、使用状況の監視装置１００５におけるバスアクセスの状態レジスタ１１０２の値を読み取り、バスアクセスの情報を動的に判定するプログラムのことである。また、置換処理とは、処理識別手段１３０２によりバス１００２を使用する処理であると識別された処理を、処理識別手段１３０２によって、前記処理と等価なリソースを使用しない等価処理に置き換えることである。

ＦＩＧ．６は、前記バスアクセスの識別装置１１０１が識別するバスアクセスの状態を示す図である。ＦＩＧ．７は、前記処理付加手段１３０３によるソフトウェアの変更を示す図である。

以下、本実施の形態のソフトウェア処理システムの動作について説明する。

ＣＰＵ１００１によって、プログラム１２０１に記述された処理１２０２，１２０３，１２０４が、この記載順に処理される。この場合に、処理１２０２，１２０３はバス１００２を使用するものとする。

ＦＩＧ．４、ＦＩＧ．５に示すように、まず、コンパイル手段１３０１は、プログラム１２０１のコンパイルの際に、処理１２０２，１２０３，１２０４をコンパイルする。同時に、処理識別手段１３０２は、処理１２０２，１２０３について、これらがバス１００２を使用する処理であると識別する。

次いで、処理付加手段１３０３は、処理１２０２と処理１２０３の実行の直前に、使用状況の判定処理および置換処理を付加する。この付加処理により、プログラム１２０１の処理は、処理１２０２の実行の直前に、使用状況の判定処理を行い、必要に応じて置換処理を実行することとなる。

バスアクセスの識別装置１１０１は、バスアクセスの頻度αを算出し、その結果をバスアクセスの状態レジスタ１１０２に記録する。このバスアクセス頻度αは、ＦＩＧ．６に示すように、バスアクセスが発生していれば増加し、バスアクセスが発生していなければ減少する。バスアクセス頻度αの増減の度合いは任意に設定できる。

ここで、バスアクセス頻度αの増減の度合いをβとする。バスアクセス頻度αは次のように算出される。まず、αに１サイクル毎にβを掛けて、次に、その値（α×β）に、バスアクセスが発生していれば１を、バスアクセスが発生してい

なければ０を足す。つまり、ｎサイクル目でのαの値をα（ｎ）とすると、

α（ｎ）＝β×α（ｎ－１）＋（１または０）

とする。例えば、βを０．９とした場合、αの値は０から１０の間の値をとることになる。

$$\alpha(1) = 1$$

$$\alpha(2) = 0.9 + 1$$

$$\alpha(3) = 0.9^2 + 0.9 + 1$$

$$\alpha(4) = 0.9^3 + 0.9^2 + 0.9 + 1$$

……

$$\alpha(n) = 0.9^{n-1} + 0.9^{n-2} + \cdots 0.9^2 + 0.9 + 1 \quad \cdots\cdots[1]$$

$$0.9 \cdot \alpha(n) = 0.9^n + 0.9^{n-1} + \cdots 0.9^2 + 0.9 \quad \cdots\cdots\cdots[2]$$

ｎが充分に大きいときは、［１］式から［２］式を減算して、

$(1 - 0.9) \cdot \alpha(n) = 1$

$\therefore \quad \alpha(n) = 10$

バスアクセスが頻繁に発生しているかどうかの目安であるバスアクセス頻度αは、極限値１０に収束することが分かる。

βを０．９とした場合のバスアクセス頻度αとサイクル数の関係はＦＩＧ．６のように示される。バスアクセス頻度αが増加している領域ではバスサイクルが頻繁に発生しており、バスアクセス頻度αが減少している領域ではバスアクセスが発生していない。

以上のように、バスアクセスの識別装置１１０１は、バスアクセス頻度αの値を算出し、その結果をバスアクセスの状態レジスタ１１０２へ記録する。

処理付加手段１３０３における使用状況の判定処理は、バスアクセスの状態レジスタ１１０２に記録されているバスアクセス頻度αを読み取り、バスアクセスが頻繁に発生しているかどうかを判定する特定の閾値と比較してバスアクセスの状態を判定する。この閾値は任意に設定することができる。前述のβ＝０．９の場合、αは０から１０の値をとることになる。閾値を５と設定したとすると、バスアクセス頻度αが５より大きければ、その時点でバスアクセスが頻繁に発生していると判定できる。逆に、バスアクセス頻度αが５以下であれば、バスアクセスの発生は頻繁でないと判定できる。

この判定の結果、バス１００２が頻繁に使用されていれば、置換処理によって処理１２０２の代わりに処理１２０５を実行する。逆に、判定の結果、バス１００２の使用が頻繁でなければ、処理１２０２をそのまま実行する。

処理１２０３についても同様に、使用状況の判定処理および置換処理を実行し、バス１００２が頻繁に使用されていれば、処理１２０３の代わりに処理１２０６を実行し、バス１００２の使用が頻繁でなければ、処理１２０３をそのまま実行する。

以上のように本実施の形態によれば、使用状況の判定処理および置換処理によって、バスアクセス競合を動的に判定し、バスアクセス頻度がある程度以上に高い場合には、バスを使用しない等価な処理に適応的に変更する。これにより、バスに対するアクセス競合の発生を軽減し、処理の停止を避けてバスを効率良く使用することが可能となり、システムの処理速度の低下の影響を軽減することができる。

なお、本実施の形態では、プロセッサを１つのＣＰＵ、リソースを１つのバスとしたが、それぞれ複数のＣＰＵ、バスでも良い。要するに、少なくとも１つのリソースと、そのリソースを使用する少なくとも１つのプロセッサであれば良い。例えば、ＣＰＵが共有メモリを使用する場合のメモリ競合についても本実施の形態によりメモリ競合の発生を軽減し、メモリを効率良く使用することが可能となり、メモリ競合の発生によるシステムの処理速度の低下の影響を軽減することができる。

なお、本実施の形態では、バスアクセスの状態レジスタにバスアクセス頻度αを求めているが、バスアクセス頻度の求め方あるいはバスアクセス頻度を求める方法については、本方法によらずともよく、指標として求めることができるものであれば同等の効果を得られる。

（第２の実施の形態）

次に、プロセッサがＤＳＰを突き放し制御方式で制御する場合の第２の実施の形態におけるソフトウェア処理システムをＦＩＧ．８～ＦＩＧ．１５を用いて具体的に説明する。突き放し制御方式では、プロセッサがＤＳＰに対して処理の要求を行い、ＤＳＰは自己の状態のいかんにかかわりなく処理要求を受け付け、プ

ロセッサは別の処理の実行に移行する。処理要求を受け付けたＤＳＰは、自己の状態を判断し、別の処理を実行中であれば待機し、前記別の処理の実行が完了した後に処理の実行を開始し、処理結果をプロセッサへ返す。この制御方式では、複数の処理がＤＳＰで競合することから、ＤＳＰが共有リソースとなっている。

ＦＩＧ．８は、第２の実施の形態にかかわるソフトウェア処理システムのハードウェアの構成を示すブロック図である。ソフトウェア処理システムは、プロセッサ２００１と、ＤＳＰ２００２と、プロセッサ２００１とＤＳＰ２００２で実行するソフトウェアを格納した記憶装置２００３と、周辺回路２００４と、バス調停回路２００５と、外部メモリ２００６と、共有バス２００７で構成される。

プロセッサ２００１はＤＳＰ２００２を突き放し制御方式で制御する。プロセッサ２００１が記憶装置２００３から読み出したソフトウェアがＤＳＰ２００２で実行すべき処理の場合は、プロセッサ２００１はＤＳＰ２００２へ処理要求を発行し、ＤＳＰ２００２は処理要求を受けた後、処理を実行する。外部メモリ２００６はプロセッサ２００１、ＤＳＰ２００２が処理を実行する際に使用する共有メモリである。バス調停回路２００５はバスアクセスを管理しており、共有メモリへのアクセス競合、すなわち共有バス２００７のバス競合が発生した場合には、外部メモリ２００６へアクセスする処理を実行するか否かを判定する。

ここで、バス調停回路２００５の制御フローをＦＩＧ．９に示す。２１０１は、共有メモリである外部メモリ２００６へのアクセス要求（読み出しまたは書き込み要求）が発生したか否かを判定するアクセス要求の判定手段である。２１０２は、現在、外部メモリ２００６がアクセス中か否かを判定するメモリ状態の判定手段である。２１０３は、アクセス要求に対しバスを解放し、メモリアクセスを促すメモリアクセスの許可手段である。２１０４は、処理が終了したか否かを判定する処理終了の判定手段である。

外部メモリ２００６へのアクセス中に新たな外部メモリ２００６へのアクセス要求が発生した場合、つまりバス競合が発生した場合、バス調停回路２００５は、一旦、アクセス要求を受理し、実際の処理に関しては先の処理が終了するまで待機（wait）状態となる。外部メモリ２００６へのアクセスが頻繁に発生すると、バス競合が発生し、バス調停回路２００５により処理が待機（wait）状態となる。その結果、ソフトウェア処理システム全体の処理速度が低下する。そこで本

実施の形態では、バス競合を避け、処理速度の低下を防ぐ。

本実施の形態におけるコンパイラは、処理記述言語で記述されたソフトウェアからオブジェクトコードを出力する通常のコンパイラとしての機能の他に、ＦＩＧ．１０の処理フローに示す機能を有する。

コンパイラは、まず、処理識別手段２２０１として外部メモリ２００６へ頻繁にアクセスする処理であるか否かを識別する。これは、外部メモリ２００６へアクセスする処理を処理識別手段２２０１があらかじめ認識していれば可能である。

次に、前記識別したアクセス高頻度の処理に対して、これと等価であるが、外部メモリ２００６へのアクセスが少ない処理である等価処理があらかじめライブラリ等で用意されているか否かを判定する。用意されていれば、コンパイラは、前記識別したアクセス高頻度の処理に対して、等価処理、記憶処理、使用状況の判定処理、置換処理を追加する。

ここで、記憶処理とは、外部メモリ２００６へ頻繁にアクセスする処理の要求が発行されてから処理が終了するまでの処理時間を記録する処理である。また、使用状況の判定処理とは、記憶処理で記憶した使用状況の情報からバス競合が発生しているか否かを判定する処理である。また、置換処理とは、前記識別したアクセス高頻度の処理を等価処理へ置き換える処理である。

ＦＩＧ．１１は、ＦＩＧ．１０の処理フローをもつコンパイラを使用したときの、頻繁にアクセスする処理の等価処理への変更の動作を示す図である（左の数字は行番号）。ＦＩＧ．１１Ａはコンパイラ実行前の処理を示し、ＦＩＧ．１１Ｂはコンパイラ実行後の処理を示す。ここでは、外部メモリ２００６へ頻繁にアクセスする処理を関数“mem_acs_many()”とし、それの等価処理を“mem_acs_few()”としている。ＦＩＧ．１２は、本実施の形態によるバス競合の判定フローを示す。

コンパイラは、あらかじめ外部メモリ２００６へアクセスする処理を登録することで、処理識別手段２２０１を用いてソフトウェアの中から関数“mem_acs_many()”を識別する。

次に、コンパイラは、前記識別したアクセス高頻度の処理に対して、上記の等価処理、記憶処理、使用状況の判定処理、置換処理を付加する。具体的には次のとおりである。

まず、ＦＩＧ．１１Ｂの２行目では、ソフトウェアに使用状況の判定処理を付加する。すなわち、“if(task_time<DEFINED_TIME)”を追加する。ここで、“task_time”は、プロセッサ２００１が外部メモリ２００６へアクセス要求を発行してから処理が終了するまでの処理時間である。また、“DEFINED_TIME”は、あらかじめ設定された設定値である。なお、設定値については、ソフトウェア処理システムの仕様に応じて的確な値を設定するのが望ましい。

外部メモリ２００６に対するアクセスの処理を実行する前に、“task_time”を参照し、“DEFINED_TIME”と比較することでバス競合を判定する。

次に、ＦＩＧ．１１Ｂの３行目から６行目は、使用状況の判定処理によりバス競合が発生していないと推定された場合の記憶処理を示す。変数“start_time”に現在時刻の“timer_count”を代入する。外部メモリ２００６に対する関数“mem_acs_many()”を実行する。変数“end_time”に現在時刻の“timer_count”を代入する。そして、“mem_acs_many()”実行の前後のタイマーの値を参照し、その差を“task_time=end_time-start_time”のように算出し、“task_time”を得る。

ＦＩＧ．１１Ｂの８行目から１０行目は、置換処理を示す。使用状況の判定処理によりバス競合が発生していると推定された場合、さらにバス競合を発生させることを防ぐため、外部メモリ２００６へのアクセスが少ない等価処理に置き換える。すなわち、“mem_acs_many()”の等価処理である“mem_acs_few()”を実行する。この置換処理で置換する処理は、処理の整合性を保つために機能的に等価であるが、外部メモリ２００６へのアクセスが少ないものが望ましい。

本実施の形態では、上述のコンパイラが追加した等価処理、記憶処理、使用状況の判定処理、置換処理により、使用状況の監視処理の過程およびソフトウェア処理の変更処理の過程を実現する。つまり、使用状況の監視処理の過程は、記憶処理により“task_time”を記憶し、使用状況の判定処理へ“task_time”を出力するものに相当する。また、ソフトウェア処理の変更処理の過程は、“task_time”に基づいて前記識別したアクセス高頻度の処理を等価処理へ置き換える置換処理に相当する。

以上のように本実施の形態によれば、“task_time”と“DEFINED_TIME”の比較結果より、バス競合が発生中か否かを判定し、バス競合が発生中と判定した場合は、

外部メモリへ頻繁にアクセスする処理を外部メモリへのアクセスが少ない処理に自動で置換する。その結果、動的にバス競合を判定しながら自動で処理の割り当てを行うことができるとともに、新たなバス競合を避けることができ、処理速度の低下を防止できる。

なお、本実施の形態における上記の発明内容を後述の発明と区別するために第１の手法とする。

ところで、第１の手法では、“task_time≧DEFINED_TIME”のときに常に等価処理への置き換えを行うようになっている。そのため、実際にはバス競合が解消されたときにも、使用状況の判定処理が、引き続きバス競合となっているという間違った判断をする可能性がある。これを避けるのが、次に説明する第２の手法である。

第２の手法としての使用状況の判定処理は、バス調停回路２００５から出力された競合情報である“task_time”と“DEFINED_TIME”との比較結果に加えて、乱数を使用することでバス競合の判定を行う。

ＦＩＧ．１３は、“task_time”と“DEFINED_TIME”の比に対する等価処理への置き換えを行う確率を示している。“task_time”と“DEFINED_TIME”の比が大きくなるに従って、等価処理へ置き換える確率が小さくなっている。

“task_time”が“DEFINED_TIME”よりも小さく、その比が１より小さい場合、使用状況の判定処理は第１の手法と同様にバス競合が発生していないと判定し、等価処理の置き換えを行わない。

しかし、“task_time”が“DEFINED_TIME”以上で、その比が１以上の場合、ＦＩＧ．１３の確率に従って等価処理への置き換えを行う。あるいは、等価処理への置き換えは行わずに、処理時間を再計測し記憶する。

第１の手法では、“task_time”が“DEFINED_TIME”以上の場合、必ずバス競合が発生中と判定していた。これに対して、乱数を用いた確率に応じてバス競合を判定する第２の手法では、第１の手法とは逆の、バス競合が発生していないという判定結果を出力する可能性を持つ。そして、ＦＩＧ．１２に示すバス競合判定フローに従い、処理時間を再度記録する。つまり、バス競合の判定に乱数を加えることで判定を見直しする。これにより、実際にはバス競合が解消されたにもかかわらず、バス競合がそのまま継続されているといった使用状況の判定処理は間違

った判断を避けることができる。

第２の手法は一連の処理の中でバス競合が頻繁に発生するシステムに対して有効である。“task_time”と“DEFINED_TIME”の比が大きいほど等価処理へ置き換える確率を小さくしているのは、“task_time”が長いほど近い未来にバス競合が解消される可能性が高いからである。

なお、ＦＩＧ．１３の確率の値については、対象システムの仕様に応じて適当な値を与える必要がある。

以上、第２の手法により使用状況の判定処理におけるバス競合の判定精度を向上できる。

なお、第２の手法では、バス競合の判定について乱数を用いた確率を使用したが、乱数を用いないバス競合の判定も考えられる。例えば、バス競合時における処理時間の平均値または最大値などの定数があらかじめ既知であれば、バス競合中と判定されてから一定周期ごとに処理時間の再計測及び記録を行うことも可能である。

次に、第３の手法について説明する。

外部メモリ２００６へ頻繁にアクセスする処理が複数の箇所に存在する場合、その処理の呼び出し元（出現箇所）別にバス競合を判定すれば判定精度は向上する。第３の方法では、処理識別手段２２０１で前記識別したアクセス高頻度の処理の出現箇所毎に処理時間を記憶し、それを使用することでバス競合の判定を高精度に行う。

ＦＩＧ．８のソフトウェア処理システムでは、複数の処理が記述された関数を実行する際には、現在のプログラムカウンタを退避レジスタに一時的に退避させた後、関数の処理を実行する。関数の処理が終了した後、退避レジスタからプログラムカウンタを読み出してプログラムカウンタを元の値に戻す。第３の手法では、出現箇所の識別処理が上述の退避レジスタからプログラムカウンタを読み出すことで処理識別手段２２０１で前記識別したアクセス高頻度の処理の出現箇所を特定する。

ＦＩＧ．１２の判定フローをもつ第１の手法に、さらに出現箇所の識別処理を追加した場合のバス競合の判定フローをＦＩＧ．１４に示す。

まず、処理識別手段２２０１で前記識別したアクセス高頻度の処理に対し、出

現箇所の識別処理により処理の出現箇所を調査する。上述したように退避レジスタを参照すれば、出現箇所を特定できる。

次に、出現箇所毎に記憶した処理時間を参照する。次に、出現箇所毎に記憶した処理時間を設定値と比較することでバス競合が発生中か否かを判定し、バス競合が発生中であれば等価処理に置き換えて処理を実行する。バス競合が発生していなければ、処理時間を記憶しながら前記識別したアクセス高頻度の処理をそのまま実行する。なお、出現箇所毎に処理時間を記憶する部分に処理時間が記憶されていない場合は、バス競合が発生していないと判定されたとみなす。この場合、処理時間を計測しながら処理識別手段２２０１で前記識別したアクセス高頻度の処理を実行し、計測した処理時間を新規に記憶する。

第３の手法は、ループ処理などの、同一の処理を繰り返して処理する場合に、出現箇所別に判定した処理時間を参照することでバス競合の判定精度が向上する。

なお、本実施の形態における“task_time”を、第１の実施の形態の使用状況の監視装置に記憶することでバス競合の判定を行うことも可能である。このときのバス競合の判定フローをＦＩＧ．１５に示す。処理時間を求めた後、使用状況の監視装置にも処理時間を記憶することにより、バス競合を動的に判定でき、バス競合に伴う処理速度の低下を防ぐことが可能である。

（第３の実施の形態）

ＦＩＧ．１６は第３の実施の形態にかかわるソフトウェア処理システムのハードウェアの構成を示すブロック図である。ソフトウェア処理システムは、ＤＳＰ４００１とプロセッサ４００２と、ＤＳＰの使用状況の監視装置４００３と、メモリ４００４で構成される。メモリ４００４は、ＤＳＰ４００１とプロセッサ４００２で実行する処理としてＦＩＧ．１７のオブジェクトコード４１０５を格納する。メモリ４００４は、ＤＳＰ４００１とプロセッサ４００２が処理を実行する際の共有メモリとして、処理結果や処理データを記憶する。

プロセッサ４００２はＤＳＰ４００１を突き放し制御方式で制御する。プロセッサ４００２がメモリ４００４から読み出したソフトウェアがＤＳＰ４００１で実行する処理の場合、プロセッサ４００２はＤＳＰ４００１へ処理要求を発行し、ＤＳＰ４００１は処理要求を受けた後、処理を実行する。ＤＳＰの使用状況の監

視装置４００３は、現在、ＤＳＰ４００１で処理を実行中か否かを記憶したレジスタ（記憶装置）であり、特定のアドレスが割り当てられているため、ソフトウェアの処理から読み出し可能である。

ソフトウェアのコンパイラのフローであるＦＩＧ．１７において、Ｃ言語などの処理記述言語で書かれたソースコード４１０２は、プロセッサ用のライブラリ４１０３とＤＳＰ用のライブラリ４１０４を用いて、オブジェクトコード４１０５に変換される。その結果、オブジェクトコード４１０５には実行に必要なプロセッサ用のライブラリ４１０６とＤＳＰ用のライブラリ４１０７が含まれる。

コンパイラは、処理記述言語からオブジェクトコードを生成する通常のコンパイラの機能の他に、ソースコード４１０２からＤＳＰ４００１で実行する処理を識別する処理識別手段と、処理識別手段で識別した処理に対し特定の処理を付加する処理付加手段を備える。

処理識別手段および処理付加手段による処理の変更をＦＩＧ．１８に示す。ここで、ＦＩＧ．１８Ａの処理記述言語で書かれたソースコードは、処理識別手段を用いて、ソースコード４１０２からＤＳＰ４００１で実行されるとして識別された処理を表している。つまり、関数“func1()”、“func2()”、“func3()”は全てＤＳＰ４００１で実行される処理である。なお、ＤＳＰ４００１で実行する処理をあらかじめ処理識別手段に登録することで、ソースコードからＤＳＰ４００１で実行する処理を識別可能である。

ＦＩＧ．１８Ｂのソースコードは、処理付加手段で付加された処理を示す。なお、処理付加手段は、func1，func3についてもfunc2と同様に特定の処理が付加される。

処理付加手段により付加される特定の処理とは、使用状況の判定処理と置換処理である。使用状況の判定処理とは、ＤＳＰ４００１が処理を実行中か否かを、ＤＳＰの使用状況の監視装置４００３を読み出すことで判定する処理である。ＦＩＧ．１８Ｂでは、８行目の“if（ＤＳＰが待ち状態）”が使用状況の判定処理に相当する。ここで待ち状態とは、ＤＳＰ４００１は処理を実行しておらず、プロセッサ４００２からの処理要求を待っている状態を表す。ＦＩＧ．１８Ｂでは９行目から１１行目では、func2の処理内容として、ＤＳＰ用のライブラリ４１０４を使用した場合を“func2_dsp()”、プロセッサ用のライブラリ４１０３を使用

した場合を“func2_cpu()”としている。これらは同等の機能をもつ処理である。

つまり、ＤＳＰ４００１を使用する処理であるか否かを識別し、その処理を実行する前にＤＳＰの使用状況の監視装置４００３を観測することでＤＳＰ４００１の状態を判定する。そして、判定結果を受けて、ＤＳＰ４００１が待ち状態であれば、そのままＤＳＰ４００１で処理を実行する。しかし、ＤＳＰ４００１が他の処理を実行中の場合は、ＤＳＰ４００１を使用しない他の処理に置換する。このようなソフトウェア処理の変更処理により、ＤＳＰ競合を避けることができる。参考として、ＦＩＧ．１９にＤＳＰ競合の判定フローを示す。

次に、処理の流れの具体例をＦＩＧ．２０を用いて説明する。

ＦＩＧ．２０Ａは、ＤＳＰ競合が発生しない場合における処理の流れである。ソフトウェアには、func0，func1，func2，func3，func4，func5，func6の処理がある。func0，func2，func4については、プロセッサ４００２からＤＳＰ４００１へ処理要求を発行し、それぞれＤＳＰ用のライブラリ４１０４を用いてfunc0_dsp，func2_dsp，func4_dspをＤＳＰ４００１上で実行する。ＤＳＰ４００１とプロセッサ４００２は共有メモリを用いて互いの処理結果を受け渡す。func0_dspとfunc2_dspとfunc4_dspとfunc5は処理結果を共有メモリ４００４に格納する。格納された処理結果は、func3とfunc6が処理を実行する際に読み出す。

ＦＩＧ．２０ＡのようにＤＳＰ競合が発生しない場合には、プロセッサ４００２とＤＳＰ４００１の資源を無駄なく使用しながら処理を実行する。

しかし、ＦＩＧ．２０Ｂのようにfunc0_dspの処理時間が長くなり、それが終了する前にfunc1がfunc2_dspの処理要求を発行した場合、ＤＳＰ競合が発生する。この場合、func0_dspが終了するまでfunc2_dspは処理を開始できない。その結果、func2_dspの処理結果を用いるfunc3も実行できなくなり、処理全体の速度が遅くなってしまう。

そこで本実施の形態では、func1がfunc2_dspの処理要求を発行した際に、ＤＳＰの使用状況の監視装置４００３を観測する。その結果、ＤＳＰ競合を認識すれば、func2と等価な機能でありＤＳＰ４００１でなくプロセッサ４００２上で処理を行うfunc2_cpuをfunc2_dspの代わりに実行する。つまり、ＤＳＰ４００１で実行する処理の処理要求を発行する際にＤＳＰ４００１の状態を監視し、処

理をＤＳＰ４００１かプロセッサ４００２に的確に割り振るソフトウェア処理の変更処理を用いる。これによりＤＳＰ競合を防ぐことができ、処理速度の低下を防ぐ。さらに、処理を的確に割り当てて待ち状態のプロセッサ４００２を使用することで、資源を効率的に使用することができる。

なお、本実施の形態のコンパイラの処理識別手段、処理付加手段により、処理サイクル数は増えるが、ＤＳＰの使用状況の監視装置４００３以外、新規にハードウェアを追加することなく、ＤＳＰ競合を避け、処理速度の低下を防止できる。

（第４の実施の形態）

第４の実施の形態によるソフトウェア処理システムをＦＩＧ．２１～ＦＩＧ．２８を用いて具体的に説明する。

ＦＩＧ．２１は、ソフトウェア処理システムのハードウェアの構成を示すブロック図である。ソフトウェア処理システムは、プロセッサ５００１と、ＤＳＰ５００２と、プロセッサ５００１とＤＳＰ５００２で実行するソフトウェアを格納した記憶装置５００３と、周辺回路５００４と、バス調停回路５００７と、外部メモリ５００５と、共有バス５００６と、調停回路５００８と、使用状況の監視装置５００９で構成される。

プロセッサ５００１はＤＳＰ５００２を突き放し制御方式で制御する。プロセッサ５００１が記憶装置５００３から読み出したソフトウェアがＤＳＰ５００２で実行すべき処理の場合は、プロセッサ５００１はＤＳＰ５００２へ処理要求を発行し、ＤＳＰ５００２は処理要求を受けた後、処理を実行する。外部メモリ５００５はプロセッサ５００１、ＤＳＰ５００２が処理を実行する際に使用する共有メモリである。ＤＳＰ５００２はバス調停回路５００８を内蔵している。調停回路５００８は、ＤＳＰ５００２で処理を実行中に新たな処理要求が発行された場合、すなわちＤＳＰ競合が発生した場合に、新たに発行された処理が実行できるか否かを判定する。また、使用状況の監視装置５００９はＤＳＰ５００２の使用状況を監視し、要求に応じて使用状況を出力する。

ここで、調停回路５００８の制御フローをＦＩＧ．２２に示す。５１０１は、ＤＳＰ５００２で実行する処理の処理要求が発生したか否かを判定する処理要求の判定手段である。５１０２は、現在、ＤＳＰ５００２がプロセッサ５００１か

らの処理要求を待っている待ち状態か否かを判定するＤＳＰ状態の判定手段である。５１０３は、処理要求に対しＤＳＰ５００２での処理の実行を促すＤＳＰ使用の許可手段である。５１０４は、処理が終了したか否かを判定する処理終了の判定手段である。

ＤＳＰ５００２で処理を実行中に新たなＤＳＰ５００２への処理要求が発生した場合、つまり、ＤＳＰ競合が発生した場合、調停回路５００８は、一旦、処理要求を受理し、実際の処理に関しては先の処理が終了するまで待機（wait）状態となる。ＤＳＰ５００２で実行する処理が頻繁に発生すると、ＤＳＰ競合が発生し、調停回路５００８により処理が待機（wait）状態となる。その結果、ソフトウェア処理システム全体の処理速度が低下する。そこで本実施の形態では、ＤＳＰ競合を避け、処理速度の低下を防ぐ。

ＦＩＧ．２３に示すように、コンパイラは、まず、処理識別手段５２０１としてＤＳＰ５００２で実行する処理か否かを識別する。これは、処理識別手段５２０１がＤＳＰで実行する処理をあらかじめ認識していれば可能である。

次に、ＤＳＰで実行する処理であると識別した処理に対して、これと機能的には等価であるが、プロセッサ５００１で実行する処理である等価処理があらかじめライブラリ等で用意されているか否かを判定する。用意されていれば、コンパイラは、前記ＤＳＰで実行する処理であると識別した処理に対して等価処理、記憶処理、使用状況の判定処理、置換処理を追加する。

ここで、記憶処理とは、ＤＳＰ５００２で実行処理の処理要求が発行されてから処理が開始するまでの待ち時間を記録する処理である。また、使用状況の判定処理とは、記憶処理で記憶した使用状況の情報からＤＳＰ競合が発生しているか否かを判定する処理である。また、置換処理とは、前記ＤＳＰで実行する処理であると識別した処理を等価処理へ置き換える処理である。

ＦＩＧ．２４は、ＦＩＧ．２３の処理フローをもつコンパイラを使用したときの、ＤＳＰ５００２で実行する処理の等価処理への変更の動作を示す図である。ＦＩＧ．２４Ａはコンパイラ実行前の処理を示し、ＦＩＧ．２４Ｂはコンパイラ実行後の処理を示す。ここでは、ＤＳＰ５００２で実行する処理を関数“dsp_task()”とし、それの等価処理を“proc_dsptask()”としている。ＦＩＧ．２５は、本実施の形態によるＤＳＰ競合の判定フローをを示す。

コンパイラは、あらかじめＤＳＰ５００２で実行する処理を登録することで、処理識別手段５２０１を用いてソフトウェアの中から関数“dsp_task()”を識別する。

次に、コンパイラは、前記ＤＳＰで実行する処理であると識別した処理に対して、上記の等価処理、記憶処理、使用状況の判定処理、置換処理を付加する。具体的には次のとおりである。

まず、ＦＩＧ．２４Ｂの２行目では、ソフトウェアに使用状況の判定処理を付加する。すなわち、“if(wait_time<DEFINED_TIME)”を追加する。ここで“wait_time”は、プロセッサ５００１がＤＳＰ５００２で実行する処理の処理要求を発行してから処理が開始するまでの待ち時間である。また、“DEFINED_TIME”は、あらかじめ設定された設定値である。なお、設定値については、ソフトウェア処理システムの仕様に応じて的確な値を設定するのが望ましい。

待ち時間“wait_time”は、過去に発生したＤＳＰ５００２で実行する処理の待ち時間を表している。現時刻に発生するＤＳＰ５００２で実行する処理要求を発行する前に“wait_time”を参照し、“DEFINED_TIME”と比較することでＤＳＰ競合を判定する。

次に、ＦＩＧ．２４Ｂの３行目から６行目は、使用状況の判定処理によりＤＳＰ競合が発生していないと推定された場合の記憶処理を示す。変数“start_time”に現在時刻の“timer_count”を代入する。ＤＳＰが処理を開始できるまで待つ。変数“end_time”に現在時刻の“timer_count”を代入する。そして、処理要求を発行した時点と“dsp_task()”の実行が開始される時点のタイマーの値を参照し、その差を“wait_time=end_time-start_time”のように算出し、“wait_time”を得る。

ＦＩＧ．２４Ｂの９行目から１１行目は、置換処理を示す。使用状況の判定処理によりＤＳＰ競合が発生していると推定された場合、さらに、ＤＳＰ競合を発生させることを防ぐため、プロセッサ５００１で実行する等価処理に置き換える。すなわち、“dsp_task()”の等価処理である“proc_dsptask()”を実行する。この置換処理で置換する処理は、処理の整合性を保つために機能的に等価であるが、ＤＳＰ５００２で実行しない処理が望ましい。

本実施の形態では、上述のコンパイラが追加した等価処理、記憶処理、使用状況の判定処理、置換処理により、使用状況の監視処理の過程およびソフトウェア処理の変更処理の過程を実現する。つまり、使用状況の監視処理の過程は、記憶処理により“wait_time”を記憶し、使用状況の判定処理へ“wait_time”を出力するものに相当する。また、ソフトウェア処理の変更処理の過程は、“wait_time”に基づいて前記ＤＳＰで実行する処理であると識別した処理を等価処理へ置き換える置換処理に相当する。

以上のように本実施の形態によれば、“wait_time”と“DEFINED_TIME”の比較結果より、ＤＳＰ競合が発生中か否かを判定し、ＤＳＰ競合が発生中と判定した場合は、ＤＳＰ５００２で実行する処理をプロセッサ５００１で実行する処理に自動で置換する。その結果、動的にＤＳＰ競合を判定しながら自動で処理の割り当てを行うことができるとともに、新たなＤＳＰ競合を避けることができ、処理速度の低下を防止できる。

なお、本実施の形態における上記の発明内容を後述の発明と区別するために第４の手法とする。

ところで、第４の手法では、“wait_time≧DEFINED_TIME”のときに常に等価処理への置き換えを行うようになっている。そのため、実際にはＤＳＰ競合が解消されたときにも、使用状況の判定処理が、引き続きＤＳＰ競合となっているという間違った判断をする可能性がある。これを避けるのが、次に説明する第５の手法である。

第５の手法としての使用状況の判定処理は、使用状況の監視装置から出力される競合情報である“wait_time”と“DEFINED_TIME”との比較結果に加えて、乱数を使用することでＤＳＰ競合の判定を行う。

ＦＩＧ．２６は、“wait_time”と“DEFINED_TIME”の比に対する等価処理への置き換えを行う確率を示している。“wait_time”が“DEFINED_TIME”よりも小さく、その比が１より小さい場合、使用状況の判定処理は第４の手法と同様にＤＳＰ競合が発生していないと判定し、等価処理の置き換えを行わない。

しかし、“wait_time”が“DEFINED_TIME”以上で、その比が１以上の場合、ＦＩＧ．２６の確率に従って等価処理への置き換えを行う。あるいは、等価処理への置き換えは行わずに、待ち時間を再計測し記憶する。

第４の手法では、“wait_time”が“DEFINED_TIME”以上の場合、必ずＤＳＰ競合が発生中と判定していた。これに対して乱数を用いた確率に応じてＤＳＰ競合を判定する第５の手法では、第４の手法とは逆の、ＤＳＰ競合が発生していないという判定結果を出力する可能性を持つ。そして、ＦＩＧ．２５に示すＤＳＰ競合判定フローに従い、待ち時間を再度記録する。つまり、ＤＳＰ競合の判定に乱数を加えることで判定を見直しする。これにより、実際にはＤＳＰ競合が解消されたにもかかわらず、ＤＳＰ競合がそのまま継続されているといった使用状況の判定処理の間違った判断を避けることができる。

第５の手法は一連の処理の中でＤＳＰ競合が頻繁に発生するシステムに対して有効である。“wait_time”と“DEFINED_TIME”の比が大きいほど等価処理へ置き換える確率を小さくしているのは、“wait_time”が長いほど近い未来にＤＳＰ競合は解消される可能性が高いからである。

なお、ＦＩＧ．２６の確率の値については、対象システムの仕様に応じて適当な値を与える必要がある。

以上、第５の手法により使用状況の判定処理におけるＤＳＰ競合の判定精度を向上できる。

なお、第５の手法では、ＤＳＰ競合の判定について乱数を用いた確率を使用したが、乱数を用いないＤＳＰ競合の判定も考えられる。例えば、ＤＳＰ競合時における待ち時間の平均値または最大値などの定数があらかじめ既知であれば、ＤＳＰ競合中と判定されてから一定周期ごとに待ち時間の再計測及び記録を行うことも可能である。

次に、第６の手法について説明する。

ＤＳＰ５００２で実行する処理が複数の箇所に存在する場合、その処理の呼び出し元（出現箇所）別にＤＳＰ競合を判定すれば判定精度は向上する。第６の方法では、処理識別手段５２０１で前記ＤＳＰで実行する処理であると識別した処理の出現箇所毎に待ち時間を記憶し、それを使用することでＤＳＰ競合の判定を高精度に行う。

ＦＩＧ．２１のソフトウェア処理システムでは、複数の処理が記述された関数を実行する際には、現在のプログラムカウンタを退避レジスタに一時的に退避させた後、関数の処理を実行する。関数の処理が終了した後、退避レジスタからプ

ログラムカウンタを読み出してプログラムカウンタを元の値に戻す。第６の手法では、出現箇所の識別処理が上述の退避レジスタからプログラムカウンタを読み出すことで処理識別手段５２０１で前記ＤＳＰで実行する処理であると識別した処理の出現箇所を特定する。

ＦＩＧ．２５の判定フローをもつ第４の手法に、さらに出現箇所の識別処理を追加した場合のＤＳＰ競合の判定フローをＦＩＧ．２７に示す。

まず、処理識別手段５２０１で前記ＤＳＰで実行する処理であると識別された処理に対し、出現箇所の識別処理により処理の出現箇所を調査する。上述したように退避レジスタを参照すれば、出現箇所の特定できる。

次に、出現箇所毎に記憶した待ち時間を参照する。次に、出現箇所毎に記憶した待ち時間を設定値と比較することでＤＳＰ競合が発生中か否かを判定し、ＤＳＰ競合が発生中であれば等価処理に置き換えて処理を実行する。ＤＳＰ競合が発生していなければ、待ち時間を記憶しながら前記ＤＳＰで実行する処理であると識別した処理をそのまま実行する。なお、出現箇所毎に記憶した待ち時間に待ち時間が記憶されていない場合は、ＤＳＰ競合が発生していないと判定されたとみなす。この場合、待ち時間を計測しながら処理識別手段５２０１で前記ＤＳＰで実行する処理であると識別された処理を実行し、計測した待ち時間を新規に記憶する。

第６の手法は、ループ処理などの、同一の処理を繰り返して処理する場合に、出現箇所別に判定した待ち時間を参照することでＤＳＰ競合の判定精度が向上する。

なお、本実施の形態における“wait_time”を、第３の実施の形態の使用状況の監視装置に記憶することでＤＳＰ競合の判定を行うことも可能である。このときのＤＳＰ競合の判定フローをＦＩＧ．２８に示す。待ち時間を求めた後、使用状況の監視装置にも待ち時間を記憶することにより、ＤＳＰ競合を動的に判定でき、ＤＳＰ競合に伴う処理速度の低下を防ぐことが可能である。

（第５の実施の形態）

ＦＩＧ．２９は第５の実施の形態にかかわるソフトウェア処理システムのハードウェアの構成を示すブロック図である。ソフトウェア処理システムは、ＤＳＰ

７００１とプロセッサ７００２と、ＤＳＰの使用状況の監視装置７００３と、メモリ７００４で構成される。メモリ７００４は、ＤＳＰ７００１とプロセッサ７００２で実行する処理としてＦＩＧ．３０のオブジェクトコードを格納する。メモリ７００４は、ＤＳＰ７００１とプロセッサ７００２が処理を実行する際の共有メモリとして、処理結果や処理データを記憶する。

プロセッサ７００２はＤＳＰ７００１を突き放し制御方式で制御する。プロセッサ７００２はメモリ７００４からソフトウェアをフェッチ、デコードし、ＤＳＰ７００１で実行する処理の場合、プロセッサ７００２はＤＳＰ７００１へ処理要求を発行することでＤＳＰ７００１は処理を実行する。ここで、プロセッサ７００２からＤＳＰ７００１への処理要求の発行について、さらに詳しく説明する。

プロセッサ７００２は、まず、ＤＳＰ７００１にあるコマンドバッファに処理コマンドおよび処理を行うのに必要なデータを書き込む。データ書き込みの終了後、プロセッサ７００２はＤＳＰ７００１へ処理の開始要求を発行する。その要求を受けたＤＳＰ７００１は、コマンドバッファから処理コマンド、データを読み出すことで処理が実行される。つまり、プロセッサ７００２からＤＳＰ７００１への処理要求とは、ＤＳＰ７００１のコマンドバッファへの処理コマンド、データの書き込みおよび処理の開始要求の発行という一連の動作を意味する。

また、ＤＳＰ７００１の使用状況の監視装置７００３は、ＤＳＰ７００１が現在処理を実行中か否かを監視するとともに、プロセッサ７００２がＤＳＰ７００１のコマンドバッファへ処理コマンドおよびデータの書き込みを行うか否かを監視する。そして、ＤＳＰ７００１が任意の処理を実行中にプロセッサ７００２が処理要求を発行するためにＤＳＰ７００１へのコマンドバッファへの書き込みを行った場合、ＤＳＰの使用状況の監視装置７００３はプロセッサ７００２へ割り込み信号を与える。割り込み信号を受けたプロセッサ７００２は、割り込みルーチンに分岐し、プロセッサ７００２からＤＳＰ７００１への処理の開始要求の発行を止める。その結果、ＤＳＰ７００１のコマンドバッファへは処理コマンド、データが格納されているが、ＤＳＰ７００１への処理の開始要求が止められるため、ＤＳＰ７００１は新たに処理を実行しない。つまり、割り込み信号により新規のＤＳＰ７００１の処理要求がキャンセルされる。

さらに、割り込み信号を受けたプロセッサ７００２は、ＤＳＰ７００１で実行

する処理の代わりに代替処理を行う。ここで代替処理とは、ＤＳＰ７００１を使用しない処理であり、本実施の形態ではＤＳＰ７００１で実行する予定だった処理と機能的には等価で、プロセッサ７００２上で実行する処理とする。

以上の割り込み信号に対する動作を実現するために、ＤＳＰ７００１で実行する処理に対し、それと機能的に等価であるが、プロセッサ７００２で実行する処理をそれぞれ用意する。そして、ＤＳＰの使用状況の監視装置７００３による割り込み信号を受けた場合、プロセッサ７００２はＤＳＰ７００１への処理要求を止め、等価な処理をプロセッサ７００２自身で代わりに実行する。

ＦＩＧ．３０に本実施の形態のソフトウェア処理システムにおけるソフトウェアを示す。main関数の中にfunc1，func2，func3，func4，func5，func6，func7，func8があり、その中でＤＳＰ７００１で実行する処理は、func2，func4，func6である。ソフトウェアにはプロセッサ７００２上で実行する処理であるプロセッサ用のライブラリと、ＤＳＰ７００１上で実行する処理であるＤＳＰ用のライブラリが含まれている。ＦＩＧ．３０におけるfunc2，func4，func6については、それぞれfunc2_dsp，func4_dsp，func6_dspというＤＳＰ用のライブラリを用いてＤＳＰ７００１上で処理を実行できるし、func2_cpu，func4_cpu，func6_cpuというＣＰＵ用のライブラリを用いてプロセッサ７００２上でも実行可能とする。

ＦＩＧ．３１は、ＦＩＧ．３０のソフトウェアを実行した際のＤＳＰ競合時の処理フローである。先に説明したようにＤＳＰ７００１が他の処理を実行中にＤＳＰ７００１に対して新規に処理要求が発行されようとしている場合、プロセッサ７００２へ割り込み信号を与え、ＤＳＰ用のライブラリを使用する処理をＣＰＵ用のライブラリを使用する処理に置き換える。

次に、処理の流れの具体例をＦＩＧ．３２を用いて説明する。

ＦＩＧ．３２Ａは、ＤＳＰ７００１のタスクが競合しない場合を示す。プロセッサ７００２でのfunc1を実行する。プロセッサ７００２はfunc1を実行後、ＤＳＰ７００１に対しfunc2の処理要求を発行し、ＤＳＰ７００１ではfunc2_dspを実行する。一方、プロセッサ７００２はfunc3を実行する。ＤＳＰ７００１はfunc2を実行後、処理結果をメモリ７００４に書き込む。メモリ７００４はプロセッサ７００２、ＤＳＰ７００１からアクセス可能な共有メモリである。プロセ

ッサ７００２でfunc3を実行後、ＤＳＰ７００１でfunc4の処理要求を発行し、ＤＳＰ７００１でfunc4_dspを実行し、プロセッサ自身はfunc5を実行する。ＤＳＰ７００１はfunc4を実行後、処理結果をメモリ７００４に書き込む。プロセッサ７００２はfunc5を実行後、ＤＳＰ７００１でfunc6の処理要求を発行し、ＤＳＰ７００１でfunc6_dspを実行し、プロセッサ自身はfunc7を実行する。ＤＳＰ７００１はfunc6を実行後、処理結果をメモリ７００４に書き込む。プロセッサ７００２は、以上の処理結果をメモリ７００４から読み取り、func8を処理する。

ＦＩＧ．３２ＡではＤＳＰ７００１のタスクが競合しないため、プロセッサ７００２とＤＳＰ７００１の資源を効率良く使用できる。

ＦＩＧ．３２Ｂは、ＤＳＰ７００１のタスクが競合する場合を示す。ここではfunc2_dspが終了しないうちに、プロセッサ７００２からfunc4の処理要求を発行され、ＤＳＰ競合が起きている。その結果、func4の処理はfunc2_dspが終了するまで停止させられるのでシステムの処理速度が低下する。

そこで本実施の形態では、ＤＳＰの使用状況の監視装置７００３がＤＳＰ競合を判定し、func4の処理要求を発行しようとするプロセッサ７００２へ割り込み信号を発生させる。それを受けたプロセッサ７００２は、自身が実行中のfunc5を中断し、func4をＣＰＵ用のライブラリであるfunc4_cpuを用いてプロセッサ７００２上で処理し、メモリ７００４に処理結果を書き込む。その後、プロセッサ７００２は中断していたfunc5を再開する。

このように、本実施の形態のソフトウェア処理システムでは、処理を実行しながらＤＳＰ競合を判定し、処理をＤＳＰ７００１かプロセッサ７００２に的確に割り振ることによって、ＤＳＰ競合を避け、処理速度の低下を防ぐ。さらに、ＤＳＰの使用状況の監視装置７００３というハードウェアで処理することによって、割り込みルーチンを用意することを除いてソースコード４１０２に対する記述の修正、追加がないため、アセンブラやオブジェクトコードで提供された場合にも対応できる。

（第６の実施の形態）

以下、本発明の第６の実施の形態を図面に基づいて説明する。

ＦＩＧ．３３は、第６の実施の形態にかかわるソフトウェア処理システムのハードウェアの構成を示すブロック図である。８００１はＣＰＵ、８００２はＤＳＰ、８００３はバス、８００４は外部メモリ、８００５は周辺回路、８００６はＣＰＵ８００１におけるメモリ、８００７はメモリ８００６中のバンク形式であるＤＳＰ８００２用のメモリバンク、８００８はメモリ８００６中のバンク形式であるＣＰＵ８００１用のメモリバンク、８００９はＤＳＰ８００２が処理を行っているかどうかを監視する使用状況の監視装置、８０１０はＤＳＰ用のメモリバンク８００７とＣＰＵ用のメモリバンク８００８とを切り替えるバンク切り替え装置である。

ＦＩＧ．３４は、ソフトウェア処理システムのソフトウェアの構成を示す図である。８１０１はＣＰＵ８００１およびＤＳＰ８００２上で動作するプログラム、８１０２，８１０３，８１０４はプログラム８１０１中に記述され、ＤＳＰ８００２を使用するコマンドを含んでライブラリ化された処理、８１０５，８１０６，８１０７は処理８１０２，８１０３，８１０４とそれぞれ同等の機能を持つ、ＣＰＵ８００１を使用するコマンドを含んでライブラリ化された処理である。

ＦＩＧ．３５は、ソフトウェア処理システムにおけるコンパイルフローを示すフローチャートである。８２０１はプログラム８１０１をコンパイルするコンパイル手段、８２０２はＤＳＰ８００２で行われる処理を識別する処理識別手段、８２０３はＤＳＰ用のメモリバンク８００７またはＣＰＵ用のメモリバンク８００８にマッピングするマッピング手段である。

以上のように構成した本実施の形態の動作について説明する。

ＦＩＧ．３６は、メモリバンクへのソフトウェアのマッピングを示す図である。

ＤＳＰ８００２はＣＰＵ８００１からの処理要求を受付けることにより処理を実行する。ＣＰＵ８００１とＤＳＰ８００２との間の制御方式は突き放し制御方式である。プログラム８１０１に記述された処理func1，func2，func3は、ＤＳＰ８００２またはＣＰＵ８００１を使用するコマンドを含み、手順として処理func1、処理func2、処理func3の順番であって、それぞれの処理にはデータ依存がないものとする。

プログラム８１０１が、コンパイラが有するコンパイル手段８２０１によってコンパイルされる際、処理識別手段８２０２によってＤＳＰ８００２を使用する

コマンドを含んだ処理である処理func1，func2，func3を識別する。そして、ＦＩＧ．３６に示すように、メモリバンクのマッピング手段８２０３によって、ＤＳＰ用のメモリバンク８００７にＤＳＰ８００２用のライブラリ化された処理８１０２，８１０３，８１０４をマッピングする。同様に、ＣＰＵ用のメモリバンク８００８にＣＰＵ８００１用のライブラリ化された処理８１０５，８１０６，８１０７をマッピングする。このとき、処理８１０２と処理８１０５は同じアドレス番地を開始アドレスとする。処理８１０３と処理８１０６、処理８１０４と処理８１０７についても同様である。このようにコンパイラは同じ名前である別々のプロセッサ用の処理を認識してコンパイルすることが可能であるという特徴を持つ。

プログラム８１０１が処理される際、最初に処理func1が行われるとする。処理func1が呼び出され、処理func1がマッピングされている開始アドレスの命令がＣＰＵ８００１によってフェッチされた瞬間に、バンク切り替え装置８０１０によってバンク切り替えのロックが実行される。このロックの実行によって、他の処理によるバンク切り替えが実行できないようになる。また、ＤＳＰ８００２が処理を行っているかどうかの情報が使用状況の監視装置８００９によって識別されていて、その情報はバンク切り替え装置８０１０への入力信号となり、バンク切り替え装置８０１０によってメモリ８００６へバンク切り替えを実行するための要求が行われている。

まず、ＤＳＰ８００２が他の処理を行っていない場合、バンク切り替えは、表側がＤＳＰ用のメモリバンク８００７であり、裏側がＣＰＵ用のメモリバンク８００８である。先ほどのバンク切り替えのロックによって、このバンク切り替えの状態は維持される。バンク切り替えのロックが実行されている間は、バンク切り替えの表側がＤＳＰ用のメモリバンク８００７になっているので、ＣＰＵ８００１はＤＳＰ用のメモリバンク８００７から処理８１０２を取り出す。処理８１０２にはＤＳＰ８００２を使用するコマンドが含まれていて、ＣＰＵ８００１によって、このＤＳＰ８００２を使用するコマンドが発行され、ＤＳＰ８００２によって処理が行われる。ＤＳＰ８００２は突き放し制御であるので、このコマンドの処理が終了する時間を特定することはできない。処理func1の呼び出し元であるプログラム８１０１のmain関数へ戻る際、このmain関数へ戻る命令がＣＰ

Ｕ８００１によってフェッチされた瞬間に、バンク切り替え装置８０１０によってバンク切り替えのアンロックが実行される。そして、メモリ８００６へのバンク切り替え要求が受け付けられ、結果として、表側がＣＰＵ用のメモリバンク８００８となり、裏側がＤＳＰ用のメモリバンク８００７となる。

これとは逆の場合、つまり、処理func1が呼び出された際、ＤＳＰ８００２が他の処理を行っていた場合、バンク切り替えは、表側がＣＰＵ用のメモリバンク８００８であり、裏側がＤＳＰ用のメモリバンク８００７である。バンク切り替えのロックが実行されているので、このバンク切り替えの状態はアンロックが実行されるまで維持される。ＤＳＰ８００２が他の処理のよって使用されているので、バンク切り替え装置８０１０はメモリ８００６へ、バンク切り替えを表側がＣＰＵ用のメモリバンク８００８、裏側がＤＳＰ用のメモリバンク８００７となるように要求することになる。

バンク切り替えは表側がＣＰＵ用のメモリバンク８００８となっているので、ＣＰＵ８００１はＣＰＵ用のメモリバンク８００８から処理８１０５を取り出す。処理８１０５は、ＤＳＰ８００２を使用するコマンドを含んだ処理８１０２と同等の機能を持っていて、ＣＰＵ８００１を使用するコマンドを含んでいるので、ＣＰＵ８００１によって処理が行われる。処理８１０５の処理が終了し、処理func1の呼び出し元であるプログラム８１０１のmain関数へ戻る際、このmain関数へ戻る命令がＣＰＵ８００１によってフェッチされた瞬間に、バンク切り替え装置８０１０によってバンク切り替えのアンロックが実行される。そして、メモリ８００６へのバンク切り替え要求が受け付けられ、結果として、表側がＤＳＰ用のメモリバンク８００７となり、裏側がＣＰＵ用のメモリバンク８００８となる。ＣＰＵ８００１が処理８１０５を行っている間に、ＤＳＰ８００２を使用していた他の処理が終了した場合は、使用状況の監視装置８００９の出力を受けて、バンク切り替え装置８０１０が、バンク切り替えの表側をＤＳＰ用のメモリバンク８００７とし、裏側をＣＰＵ用のメモリバンク８００８とする要求を行うことになる。しかし、この要求が受け付けられてバンク切り替えが実行されるのは、前述と同様にバンク切り替えのアンロックが実行されてからである。

次に、処理func1のＤＳＰ８００２を使用する処理８１０２中のコマンドが、ＤＳＰ８００２によって処理が行われている間、次の処理である処理func2が呼

び出されたとする。このとき、ＤＳＰ８００２は使用中であり、先ほどのバンク切り替えのアンロックはバンク切り替え装置８０１０によって実行されているので、バンク切り替えは表側がＣＰＵ用のメモリバンク８００８となり、裏側がＤＳＰ用のメモリバンク８００７となっている。処理func2が呼び出され、処理func2がマッピングされている開始アドレスの命令がＣＰＵ８００１によってフェッチされた瞬間に、バンク切り替え装置８０１０によってバンク切り替えのロックが実行される。このロックの実行によって、他の処理によるバンク切り替えが実行できないようになる。

バンク切り替えは表側がＣＰＵ用のメモリバンク８００８となっているので、ＣＰＵ８００１はＣＰＵ用のメモリバンク８００８から処理８１０６を取り出す。処理８１０６は、ＤＳＰ８００２を使用するコマンドを含んだ処理８１０３と同等の機能を持っていて、ＣＰＵ８００１を使用するコマンドを含んでいるので、ＣＰＵ８００１によって処理が行われる。処理８１０６の処理が終了し、処理func2の呼び出し元であるプログラム８１０１のmain関数へ戻る際、このmain関数へ戻る命令がＣＰＵ８００１によってフェッチされた瞬間に、バンク切り替え装置８０１０によってバンク切り替えのアンロックが実行される。そして、メモリ８００６へのバンク切り替え要求が受け付けられ、結果として、表側がＤＳＰ用のメモリバンク８００７となり、裏側がＣＰＵ用のメモリバンク８００８となる。ＣＰＵ８００１が処理８１０６を行っている間に、ＤＳＰ８００２を使用していた処理８１０２中のコマンドが終了した場合は、先ほどの処理func1の処理を行う際にＤＳＰ８００２が使用中であった場合と同様に、バンク切り替え装置８０１０によってメモリ８００６へバンク切り替えの要求が行われ、バンク切り替えのアンロックが実行されてから、この要求が受け付けられることになる。

最後に処理func3が呼び出される。先ほどのＤＳＰ８００２が使用中でない場合の処理func1の場合と同様に、バンク切り替え装置８０１０によって、バンク切り替えのロック、バンク切り替え要求が行われ、ＤＳＰ用のメモリバンク８００７からＤＳＰ８００２を使用するコマンドを含んだ処理８１０４がＣＰＵ８００１によって取り出され、ＤＳＰ８００２を使用するコマンドは突き放し処理としてＤＳＰ８００２によって実行される。プログラム８１０１のmain関数へ戻る際のバンク切り替えのアンロックの実行、バンク切り替えの表側をＣＰＵ用の

メモリバンク８００８とし、裏側をＤＳＰ用のメモリバンク８００７とするバンク切り替えの実行についても同様である。

以上のように本実施の形態によれば、バンク切り替え装置８０１０を用いた方法によって、ＤＳＰ８００２におけるＤＳＰ競合の発生を軽減し、ＤＳＰ８００２を効率良く使用することが可能となる。この方法を実現するために、プログラム８１０１を修正する必要がなくそのまま適用できるので、ソフトウェアのオーバーヘッドがほとんどない。また、使用状況の監視装置８００９およびバンク切り替え装置８０１０という専用のハードウェアを用いることによって、バス８００３の使用状況の監視およびソフトウェア処理変更を比較的高速に実現することができる。

なお、本実施の形態では、プロセッサを１つのＣＰＵ、リソースを１つのＤＳＰとしたが、それぞれ複数のＣＰＵ、ＤＳＰでも良い。要するに、少なくとも１つのリソースと、そのリソースを使用する少なくとも１つのプロセッサであれば良い。

From the above description, it will be apparent that the present invention provides.

What is claimed is:

1．プロセッサが使用するリソースの使用状況を監視する使用状況の監視処理の過程と、

前記使用状況の監視処理の過程で得られる競合情報に応じて、実行されるソフトウェアの処理方法を適応的に変更するソフトウェア処理の変更処理の過程とを含むソフトウェア処理方法。

2．前記ソフトウェア処理の変更処理の過程は、ソフトウェアがある処理を行うのに複数の実行方法を持ち、前記ソフトウェアの実行中に前記使用状況の監視処理の過程で得られる競合情報に応じて、前記複数の実行方法の中から１つを選択する請求項１に記載のソフトウェア処理方法。

3．前記リソースは、処理の記憶装置であり、

前記使用状況監視処理の過程は、前記記憶装置の使用状況を監視する請求項１に記載のソフトウェア処理方法。

4．前記リソースは、処理の記憶装置であり、

前記使用状況監視処理の過程は、前記記憶装置の使用状況を監視する請求項２に記載のソフトウェア処理方法。

5．前記使用状況監視処理の過程は、複数クロック遡った記憶装置の使用状況を記憶し、過去および現在の使用状況から前記競合情報を生成する請求項３に記載のソフトウェア処理方法。

6．前記使用状況監視処理の過程は、複数クロック遡った記憶装置の使用状況を記憶し、過去および現在の使用状況から前記競合情報を生成する請求項４に記載のソフトウェア処理方法。

7．前記使用状況監視処理の過程は、前記記憶装置の使用時における使用時間を記憶し、前記使用時間が所定値以上か否かに基づき前記競合情報を生成するこ

とを特徴とする請求項３に記載のソフトウェア処理方法。

８．前記使用状況監視処理の過程は、前記記憶装置の使用時における使用時間を記憶し、前記使用時間が所定値以上か否かに基づき前記競合情報を生成することを特徴とする請求項４に記載のソフトウェア処理方法。

９．前記リソースは、処理の記憶装置、および、前記プロセッサと前記記憶装置を接続するバスであり、

前記使用状況の監視処理の過程は、前記バスの使用状況を監視する請求項１に記載のソフトウェア処理方法。

１０．前記リソースは、処理の記憶装置、および、前記プロセッサと前記記憶装置を接続するバスであり、

前記使用状況の監視処理の過程は、前記バスの使用状況を監視する請求項２に記載のソフトウェア処理方法。

１１．前記使用状況の監視処理の過程は、複数クロック遡ったバスの使用状況を記憶し、過去および現在の使用状況から前記競合情報を生成する請求項９に記載のソフトウェア処理方法。

１２．前記使用状況の監視処理の過程は、複数クロック遡ったバスの使用状況を記憶し、過去および現在の使用状況から前記競合情報を生成する請求項１０に記載のソフトウェア処理方法。

１３．前記使用状況の監視処理の過程は、前記バスの使用時における使用時間を記憶し、前記使用時間が所定値以上か否かに基づき前記競合情報を生成する請求項９に記載のソフトウェア処理方法。

１４．前記使用状況の監視処理の過程は、前記バスの使用時における使用時間を記憶し、前記使用時間が所定値以上か否かに基づき前記競合情報を生成する請

求項１０に記載のソフトウェア処理方法。

１５．前記リソースは、前記プロセッサからの処理要求に応じて処理を行う第２のプロセッサであって、

前記使用状況の監視処理の過程は、前記第２のプロセッサの使用状況の監視を行う請求項１に記載のソフトウェア処理方法。

１６．さらに、同一アドレスでアクセス可能な複数のメモリバンクを有し、前記使用状況の監視処理の過程で得られる競合情報は、前記複数のメモリバンクのうち１つのメモリバンクの選択を示す信号である請求項１５に記載のソフトウェア処理方法。

１７．さらに、ソフトウェアのコンパイラを有し、

前記コンパイラは、

ソフトウェアから前記リソースを使用する処理であるか否かを識別する処理識別手段と、

前記処理識別手段により識別された処理と等価な、前記リソースを使用しない等価処理と、

前記使用状況の監視処理の過程で得られる競合情報により使用状況を判定する使用状況の判定処理と、

前記使用状況の判定処理の結果に応じて、前記処理識別手段により識別された処理を適応的に前記等価処理に置き換える置換処理と

を、前記ソフトウェアに追加する請求項１に記載のソフトウェア処理方法。

１８．さらに、ソフトウェアのコンパイラを有し、

前記コンパイラは、

ソフトウェアから前記リソースを使用する処理であるか否かを識別する処理識別手段と、

前記処理識別手段により識別された処理と等価な、前記リソースを使用しない等価処理と、

現時刻の前記使用状況の監視処理の過程で得られる競合情報を記憶する記憶処理と、

過去の時刻における前記競合情報により使用状況を判定する使用状況の判定処理と、

前記使用状況の判定処理の結果に応じて、前記処理識別手段により識別された処理を適応的に前記等価処理に置き換える置換処理と

を、前記ソフトウェアに追加する請求項１に記載のソフトウェア処理方法。

１９．前記競合情報は、前記リソースの処理要求が発行されてから処理が終了するまでの処理時間であり、

前記使用状況の判定処理は、前記処理時間をある設定値と比較する処理である請求項１７に記載のソフトウェア処理方法。

２０．前記競合情報は、前記リソースの処理要求が発行されてから処理が終了するまでの処理時間であり、

前記使用状況の判定処理は、前記処理時間をある設定値と比較する処理である請求項１８に記載のソフトウェア処理方法。

２１．前記競合情報は、前記リソースの処理要求が発行されてから処理が実行するまでの待ち時間であり、

前記使用状況の判定処理は、前記待ち時間をある設定値と比較する処理である請求項１７に記載のソフトウェア処理方法。

２２．前記競合情報は、前記リソースの処理要求が発行されてから処理が実行するまでの待ち時間であり、

前記使用状況の判定処理は、前記待ち時間をある設定値と比較する処理である請求項１８に記載のソフトウェア処理方法。

２３．前記使用状況の判定処理は、定期的または不定期に前記リソースの使用状況の判定を見直す請求項１７に記載のソフトウェア処理方法。

２４．前記使用状況の判定処理は、定期的または不定期に前記リソースの使用状況の判定を見直す請求項１８に記載のソフトウェア処理方法。

２５．前記使用状況の判定処理は、乱数を用いて前記リソースの使用状況の判定を見直す請求項２３にソフトウェア処理方法。

２６．前記使用状況の判定処理は、乱数を用いて前記リソースの使用状況の判定を見直す請求項２４にソフトウェア処理方法。

２７．請求項１８に記載のソフトウェア処理方法において、

前記コンパイラは、前記処理識別手段により抽出される処理が前記ソフトウェアの複数の箇所から抽出される場合、さらに、前記処理識別手段で識別された処理の出現箇所を識別する出現箇所の識別処理を前記ソフトウェアに追加し、前記記憶処理は前記出現箇所毎に前記競合情報を記憶し、前記使用状況の判定処理は前記出現箇所毎に記憶した前記競合情報を用いて判定を行うソフトウェア処理方法。

２８．プロセッサと、

前記プロセッサが使用するリソースと、

前記リソースの使用状況を監視する使用状況の監視装置と、

前記使用状況の監視装置で得られる競合情報に応じて、実行されるソフトウェアの処理方法を適応的に変更するソフトウェア処理の変更装置と

を有するソフトウェア処理システム。

２９．前記ソフトウェア処理の変更装置は、ソフトウェアがある処理を行うのに複数の実行方法を持ち、前記ソフトウェアの実行中に前記競合情報に応じて、前記複数の実行方法の中から１つを選択する請求項２８に記載のソフトウェア処理システム。

３０．前記リソースは、前記プロセッサからの処理要求に応じて処理を行う第２のプロセッサであって、

前記使用状況の監視装置は、前記第２のプロセッサの使用状況の監視を行う請求項２８に記載のソフトウェア処理システム。

３１．前記競合情報は、前記プロセッサへの割り込み信号である請求項３０に記載のソフトウェア処理システム。

ABSTRACT

ＣＰＵのようなプロセッサとＤＳＰで構成され、プロセッサとＤＳＰとが外部メモリやバスを共有リソースとし、ＤＳＰはプロセッサからの処理要求に応じて処理を行うマルチプロセッサシステムにおいて、使用状況の監視処理の過程は、ＤＳＰの使用状況を監視し、前記使用状況の監視処理の過程で得られる競合情報が頻繁な使用を示しているときは、ソフトウェア処理の変更処理の過程は、実行されるソフトウェアの処理方法を適応的に変更し、等価処理に切り替えることにより、バス競合を避け、処理速度の低下を防ぐ。